ALLIGATOR: THEOREM PROVING FOR DEPENDENT TYPE SYSTEMS WITH SIGMA TYPES

Paul Piwek
Centre for Research in Computing, The Open University
Walton Hall, Milton Keynes, UK

March, 2006

Abstract This paper describes a theorem prover for Dependent Type Systems. We start with an introduction to Dependent Type Systems and highlight the properties that make them specifically suited for computational semantics. We proceed with a brief description of the ALLIGATOR system, including its architecture and some implementation issues. Alligator works with a specific generalization of Dependent Type Systems: Pure Type Systems extended with Sigma Types. The paper concludes with examples of proofs constructed by ALLIGATOR. These have been selected to illustrate how certain problems in anaphora/presupposition resolution can be addressed with the current system.

1 Introduction

Automated reasoning plays a pivotal role in computational semantics. It has a place both in the evaluation and construction of interpretations (for examples of the latter, see section 4 of this paper). Automated *symbolic* reasoning, which we focus on in this paper, requires a formal language as the substratum of the reasoning. Blackburn and Bos ([7]) make a good case for the use of First Order Predicate Logic (FOPL) in computational semantics. They see FOPL as a sensible starting point, both for practical reasons (availability of high performance theorem provers and to a lesser extent model builders) and theoretical reasons (they discuss a range of interesting phenomena which can be dealt with in FOPL). They address the criticism that FOPL lacks the 'dynamic potential' of other formalism such as Discourse Representation Theory (DRT; [18]) by suggesting that we should allow for free movement between representation formalisms.

We agree with the idea that FOPL is a good starting point, but also think that for computational semantics to develop further as a field, extensions going beyond FOPL should be actively explored. In this paper, a research tool is described that takes such explorations in one particular direction. The tool is a theorem prover for Dependent Type Systems [4, 5]. The Sicstus Prolog source code of this prover is available, free of charge, for research purposes ([21]). We explain in detail what Dependent Type Systems (DTS) are in the next section. For now, let us lay out the reasons for choosing DTS as a suitable formalism for computational semantics.

1. The notion of a *context* that is built up *incrementally* is inherent to DTS. In this respect, DTS share the 'dynamic potential' of, for instance, DRT.
2. The specific DTS that we will work with allows for a lot of flexibility regarding the power of the logic. By varying a limited number of parameters, it is possible to switch from, for example, propositional to predicate logic, or first order to higher order logics.

3. In standard logics and also DRT, the signature (specifying non-logical constants) is defined separately. In DTS, the signature is itself part of the context. It can be incrementally extended, whenever necessary. This is useful for modeling language users who encounter and learn new concepts. Such new concepts can be incrementally added to the context.
4. The underlying logic of DTS is constructive in the sense that the excluded middle ($p \vee \neg p$) does not hold. It is, however, straightforward to extend the context with this principle or weaker variants. We might only want to apply the principle to certain predicates and not others, or only in certain situations. For instance, *is red* might not behave according to the principle of the excluded middle: if I have a car which is half red and half green, then it would be wrong to say that it is either red or not red. (Things are different for the predicate *is partly red*). More generally, the context of a DTS can be used to introduce and reason with (multiple) non-standard logics. This is the 'logical framework' use of DTS. See [15] for an overview of alternative ways of 'doing logic' in DTS.
5. When one does logic in a DTS, the system records explicit representations of Gentzen-style ([14]) natural deduction proofs. This has the following advantages:

   (a) From a practical point of view, there is a lot to say for explicit representations of proofs, in particular, when one is concerned about the reliability of theorem provers. This is expressed by the *de Bruijn criterion* for reliable proof systems: "A proof assistant satisfies the de Bruijn criterion if it generates 'proof-objects' (of some form) that can be checked by an easy algorithm." (cited from [5])

   (b) The fact that DTS proofs correspond with natural deduction proofs, is also of interest if one is concerned with models of human reasoning in natural language understanding. In some schools of thought in psychology, it has been argued that natural deduction is a good approximation of human reasoning (see, e.g., [24]).

   (c) Proof objects can also help to identify proofs which are valid but spurious in the sense that they do not really consume their premises (e.g., a proof from any proposition to a necessary truth). The structure of proof objects can be used to identify inferences in which the premises are irrelevant ([17]).

   (d) Explicit proof objects provide direct access to the justifications that an agent has for the conclusions and the interpretations that it constructs. This is particularly useful for dialogue agents that need to respond to utterances of other agents. Such responses can themselves again be queried, for example, through clarificatory questions (a point made also in [25]) and why questions (A:p, B: no, $\neg p$, A: Why $\neg p$?). In order to respond appropriately, the agent needs to access its own background knowledge and how it was used for drawing conclusions. A proof object provides a compact representation of this information.

6. The DTS that we will work with have well understood meta-mathematical properties ([4]). DTS are a major topic of investigation in theoretical computer science.
7. DTS-style analyses exist for a wide range of interesting phenomena including donkey sentences ([26]), anaphoric expressions and temporal reference [23], belief revision

([9]), bridging anaphora [22], clarification ellipsis ([11]), metonymy ([10]), inter-agent communication, knowledge and observation ([1]), ontological reasoning for feedback dialogues ([6]), and human-machine dialogue ([2]). Additionally, there is research on relating DTS proof-theoretic natural language semantics to model-theoretic approaches ([12]), and there are studies employing the related formalism of labelled deduction to natural language semantics ([19]). In 2005, the 2^{nd} Workshop on Lambda-Calculus, Type Theory, and Natural Language took place at King's College London.

Certainly, there are other formalisms that have one or more of these properties. The point is, however, that, to the best of our knowledge, there are no other formalisms that bring *all* of these properties together in *one* system.

The theorem prover that we present here is distinct from other publicly available provers for DTS in that it directly constructs proof objects for natural deduction proofs. Other provers for DTS typically work with internal representations that are only at the end of the reasoning process *translated* to natural deduction proof objects. For example, COCKTAIL ([13]) uses tableaux and translates these, whereas TPS ([3]) is based on the mating method. The handbook chapter by Barendregt and Geuvers on proof assistants for DTS ([5]) lists a number of further automated theorem provers, none of which works directly with proof objects. The systems listed in [5] that do work directly with proof objects are all *interactive* programs. They require guidance from the user during the construction of a proof, something which *fully automated* theorem provers do not require.

ALLIGATOR distinguishes itself in two other ways from existing theorem provers for DTS. Firstly, as opposed to these other systems it was not developed with mathematical/program specification reasoning in mind, but rather for inferences in language interpretation. As a consequence, it has been streamlined to link up with notation and functionality relevant to computational semantics (specifically, allowing for notation which is close to DRT and omission of inductive types). Secondly, to the best of my knowledge ALLIGATOR is the only automated theorem prover which directly conforms to the specification of Pure Type Systems ([4]), the most general and flexible kind of DTS (most DTS can be emulated in PTS; see [20] for an overview of DTS and their counterparts in PTS).

The remainder of this paper contains the following sections. In Section 2, the notion of a Dependent Type System is introduced and a formal specification of such systems is given. Section 3 contains a description of the ALLIGATOR system. This is followed, in Section 4, by some examples of proofs generated by ALLIGATOR. The paper ends with a conclusions section.

2 Dependent Type Systems

DTS come in wide variety of flavours and variations. An excellent overview of their historical roots can be found in [20]. All these systems share, however, two features. Firstly, they are *type systems*. That is, given a set of assumptions Γ, also know as the *context*, they provide rules for determining whether a particular object, say a, belongs to a given type, say t. We write $\Gamma \vdash a : t$, if, given the context Γ, a is of type t, i.e., a *inhabits* type t. The objects that

are classified using type systems are terms of the λ-calculus.[1] Γ is a sequence of statements $x_1 : t_1, \ldots, x_n : t_n$ (with $n \geq 0$).

The second feature of DTS is that of *dependency*. First, there is dependency between statements in the context: in order to use a type t_k to classify an object x_k, this type t_k needs to have been introduced in that part of the context that *precedes* it. In other words, t_k can only be used if it itself inhabits a type or can be constructed from other types that are available in the context preceding it. Now, of course, it is not possible that all types need to be introduced in the context before they can be used, since that would require an infinitely long context. For that reason, there is also a, usually small, set of specifically designated types that are called *sorts*. These require no context; they can be used even in the empty context. One sort can also inhabit another sort ($s_1 : s_2$), but such statements are defined *outside* the context in the form of axioms. Furthermore, a sort does not need to inhabit another sort (that is, some sorts are at the top of the type hierarchy). Second, there is a variety of dependency that occurs *inside* types. Since type systems are used to classify terms of the λ-calculus, they can also deal with functions. A function f from objects of type t_1 to objects of type t_2 inhabits the function type $t_1 \rightarrow t_2$. *Dependent* function types are a generalization of function types: a dependent function type is a function type where the range of the function changes depending on the object to which the function is applied. The notation for dependent function types is $\Pi x : A.B$. If we apply an inhabitant of this function type, say f, to an object of type A, then the resulting object fa (f applied to a) is of type B, but with all free occurrences of x in B substituted with a (that is, the type of fa is $B[x := a]$).

Before we provide a formal description of a particular DTS, let us examine how one can do logic in a DTS.[2] From a logical point of view, we are interested in propositions as the constituents of deductive arguments. In classical logic, one focuses on judgements of the following form: the truth of proposition q follows/can be derived from the truth of the propositions $p_1, \ldots, p_n$. We reason from the truth of the premises to the truth of the conclusion. To do logic in a DTS, we move from *truth* to *proof*: we, now, reason from the proofs that we (assume to) have for the premises to a proof for the conclusion. In other words, we are interested in judgements of the following form: a is proof of proposition q follows/can be derived from assuming that a_1 is a proof of p_1, a_2 is a proof of p_2, ..., and a_n is a proof p_n. This kind of judgement can be formalized in a DTS as $a_1 : p_1, \ldots, a_n : p_n \vdash a : p$. Thus, we read $a : p$ as 'a is a proof for p'. For this purpose, we model proofs as (λ-calculus) terms and propositions as (a certain class of) types. This is known as the Curry-Howard-de Bruijn embedding. Importantly, given such an embedding, cut-elimination in the logic corresponds to reduction in the term calculus.

The embedding is grounded in the Brouwer-Heyting-Kolmogorov interpretation of proofs as *constructions*. For example, a proof for a conditional $p \rightarrow q$ is identified with a method that transforms a proof of p into a proof for q. In a DTS, this is formalized by modelling the

[1] In particular, the 'well-behaved' terms of the λ-calculus, i.e., terms that have a normal form, unlike, for example $(\lambda x.xx)(\lambda x.xx)$, which can be rewritten ad infinitum.

[2] Here we restrict our discussion to the direct encoding of logic in type theory: logical connectives have their direct counterparts in the DTS (e.g., implication is modelled in terms of function types; see below). An alternative is the 'logical framework' approach. In that approach, the language of the logic and its rules of inference are explicitly declared in the context Γ. This means that logical connectives no longer correspond directly with constructions of the type theory.

proof f for a type $p \rightarrow q$ as a function from objects of type p to objects of type q, such that if a is a proof of p (i.e., $a : p$), then f applied to a is a proof of q (i.e., $fa : q$). Universal quantification is dealt with along the same lines. A proof of the proposition $\forall x \in A : P(x)$ is identified with a method which for every object $a \in A$ returns a proof for $P(a)$. In a DTS, the counterpart for universal quantification is the dependent function type. In particular, $\forall x \in A : P(x)$ becomes $(\Pi x : A.Px)$. A proof for this type is a function f which, given an object $a : A$, returns the proof fa for Pa.

2.1 Pure Type Systems with Sigma Types

Pure Type Systems (PTS; [4]) are of particular interest, because of their generality. With the help of a small number of parameters, PTS can be tailored to match a wide variety of DTS. ALLIGATOR implements PTS$^{\Sigma}$, which is an extension of PTS with Σ types. Σ types are also known as dependent product types and can be used to model $\wedge$ and $\exists$.

To give a formal definition of PTS$^{\Sigma}$, we start by defining the basic vocabulary. We have disjoint sets V and $\mathcal{S}$ of VARIABLES and SORTS, respectively. The set of PSEUDO-TERMS T is defined as follows: $T ::= V \mid \mathcal{S} \mid TT \mid \lambda V : T.T \mid \Pi V : T.T \mid (T, T) \mid \Sigma V : T.T \mid \pi_1 T \mid \pi_2 T$.

An ASSUMPTION A of the form $V : T$ is formed using a variable and a pseudo-term. A PSEUDO-CONTEXT Γ consists of a sequence of such assumptions: $\Gamma ::= \epsilon \mid \Gamma, A$. ($\epsilon$ is the empty sequence).

The skeleton of a PTS$^{\Sigma}$ consists of a number of derivation rules that axiomatize the relation $\vdash$. To obtain a specific PTS$^{\Sigma}$ four parameters need to be fixed: $(\mathcal{S}, \mathcal{A}, \mathcal{R}_{\Pi}, \mathcal{R}_{\Sigma})$. 1. $\mathcal{S}$ is the set of sorts, 2. $\mathcal{A}$ is a set of axioms of the form $s : s'$, where $s, s' \in \mathcal{S}$, and 3. $\mathcal{R}_{\Pi} \subseteq \{(x, y) \mid x \in \mathcal{S}, y \in \mathcal{S}\}$[3], and 4. $\mathcal{R}_{\Sigma} \subseteq \{(x, y) \mid x \in \mathcal{S}, y \in \mathcal{S}\}$.

The DERIVATION RULES that axiomatize the relation $\vdash$ are specified below. The usual notions of fresh variable (Γ-fresh: a variable which does not occur in Γ) and substitution (The replacement of all occurrences of a variable in a term with another variable) are used.[4]

[3]We restrict our attention to functional or singly sorted PTS which cover most PTS and for which *uniqueness of types holds*. That is, if a term has two types, then the types are equal under reduction; see page 225 of [4].

[4]For the sake of completeness, we define the notion of equality ($=$) that is use here, after some auxiliary definitions: ONE-STEP-REDUCTION ($\triangleright_1$) is defined by the following three rules: 1. $(\lambda x : A.M) \cdot N \triangleright_1 M[x := N]$ (beta-reduction), 2. $\pi_1((A, B)) \triangleright_1 A$, and 3. $\pi_2((A, B)) \triangleright_1 B$. We then define REDUCTION as follows: $E \triangleright E'$ iff $E \triangleright_1 E'$ or $E = E'$ or $\exists E'' : E \triangleright E'' \wedge E'' \triangleright E'$. Next, we define CONGRUENCE. $P \equiv_{\alpha} Q$ iff Q has been obtained from P by a finite (perhaps empty) series of changes of bound variables (i.e. variables which are within the scope of a Π, Σ or λ. Finally, we define EQUALITY ($=$): $P = Q$ iff Q is obtained from P by a finite (perhaps empty) series of reductions and reversed reductions and changes of bound variables. More precisely: $P_0 = P_n$ iff there exist $P_1, \ldots P_{n-1}$ such that for $0 \leq i \leq n-1$: ($P_i \triangleright_1 P_{i+1}$ or $P_i \triangleleft_1 P_{i+1}$ or $P_i \equiv_{\alpha} P_{i+1}$).

(axioms)	$\epsilon \vdash s : s'$	if $s : s' \in \mathcal{A}$
(start)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$	x is $\Gamma-$fresh and $s \in \mathcal{S}$
(weaken)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$	x is $\Gamma-$fresh and $s \in \mathcal{S}$
(form)	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_2}$	$(s_1, s_2) \in \mathcal{R}_\Pi$
(intro)	$\dfrac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)}$	$s \in \mathcal{S}$
(elim)	$\dfrac{\Gamma \vdash F : (\Pi x : A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash F \cdot a : B[x := a]}$	
(conv)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B = B'}{\Gamma \vdash A : B'}$	$s \in \mathcal{S}$
(Σ-form)	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Sigma x : A.B) : s_2}$	$(s_1, s_2) \in \mathcal{R}_\Sigma$
(Σ-intro)	$\dfrac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[x := a] \quad \Gamma \vdash (\Sigma x : A.B) : s}{\Gamma \vdash (a, b) : (\Sigma x : A.B)}$	$s \in \mathcal{S}$
(π_1)	$\dfrac{\Gamma \vdash p : (\Sigma x : A.B)}{\Gamma \vdash (\pi_1 p) : A}$	
(π_2)	$\dfrac{\Gamma \vdash p : (\Sigma x : A.B)}{\Gamma \vdash (\pi_2 p) : B[x := (\pi_1 p)]}$	

2.2 Logic in PTS$^\Sigma$

Here we describe the system PTS$^\Sigma$ that ALLIGATOR comes with by default and which was also the system used for the proofs in section 4. The PTS$^\Sigma$ in question is λPREDω^Σ (an extension of λPREDω of the logic cube, see e.g. [4], with certain Σ types):

1. $\mathcal{S}$ = $\{Set, Type^{Set}, Prop, Type^{Prop}\}$
2. $\mathcal{A}$ = $\{Set : Type^{Set}, Prop : Type^{Prop}\}$
3. $\mathcal{R}_\Pi$ = $\{(Set, Set), (Set, Type^{Prop}), (Type^{Prop}, Type^{Prop}),$ $(Prop, Prop), (Set, Prop), (Type^{Prop}, Prop)\}$
4. $\mathcal{R}_\Sigma$ = $\{(Set, Prop), (Prop, Prop)\}$

Set and $Prop$ are sorts which we use to type sets and propositions, respectively. Set is itself an inhabitant of $Type^{Set}$ and $Prop$ of $Type^{Prop}$. Let us first look at the rules for Π. The rule (Set, Set) allows us to construct function types such as ($A \to A$) and ($A \to (A \to A)$).

$(Set, Type^{Prop})$ allows us to make, for example, predicates of type $A \rightarrow Prop$ (i.e., the type of a function from objects of type A into propositions) and relations with the function type $(A \rightarrow (A \rightarrow Prop))$. $(Prop, Prop)$ is for functions from proofs of propositions to proofs of proposition, i.e., logical implication. $(Set, Prop)$ is for universal quantification over sets. $(Type^{Prop}, Prop)$ enable quantification over domains of type $Type^{Prop}$, i.e., quantification over predicates and relations. Finally, $(Type^{Prop}, Type^{Prop})$ allows for quantification over $Type^{Prop}$, that is all higher order domains.

We have only two rules for Σ: $(Set, Prop)$ for existential quantification over domains of type set, and $(Prop, Prop)$ for conjunction of propositions. Note that it has been proved that addition of any member of $\{(Type^{Prop}), Prop)$, $(Type^{Prop}), Set)$, $(Type^{Set}), Prop)$, $(Type^{Set}), Set)\}$ to $\mathcal{R}_{Sigma}$ leads to inconsistency of the system (i.e., $(\Box, \star)$ in the Calculus of Constructions, see [16]).

In order to do logic, we need some further axioms. We add these to the base context (a collection of assumptions that we always use when trying to construct a proof). In particular, we introduce the false as a proposition ($false : Prop$), and the information that from the false we can derive anything: $abs : (\Pi p : prop.false \rightarrow p)$. Now we can define negation: $\bar{P} = P \rightarrow false$ and disjunction $A \vee B = (A \rightarrow false) \rightarrow B$.[5] We also occasionally might need the double negation rule (which is equivalent to adopting the excluded middle): $dn\_pr : (\Pi p : Prop.((p \rightarrow false) \rightarrow false) \rightarrow p)$. dn_pr returns for each proposition p a proof that $((p \rightarrow false) \rightarrow false)$ implies p.

3 System Architecture and Implementation

ALLIGATOR 1.0 has been implemented in Sicstus PROLOG and been tested with version 3.12.2 of Sicstus. The main predicate is:

```
prove(+Pseudo_Context,-Term:+Goal).
```

`Term` is a PROLOG variable which the system will instantiate with a proof term for `Goal`, if such a term exists (given a certain search depth). Otherwise, the call fails. Formally, `Goal` is a pseudo term.

(1) PSEUDO-TERMS (PT) are defined as follows:

```
PT ::= S | V | A | PT-PT | pair(PT,PT) | pi1(PT) |
       pi2(PT) | lambda(V:PT,PT) | pi(V:PT,PT) |
       sigma(V:PT,PT)
 V ::= Prolog Variable
 A ::= Prolog Atom
 S ::= type_prop | type_set | prop | set
```

The constructors `-, pair(x,y), pi1, pi2, lambda, pi, sigma` stand, respectively, for function application (left associative), pairs (i.e., (x,y)), left and right projection,

[5] Alternatively, on can define disjunction inside the context: $\vee : Prop \rightarrow (Prop \rightarrow Prop)$, $in_1 : \Pi p, q : Prop.p \rightarrow (p \vee q)$, $in_2 : \Pi p, q : Prop.q \rightarrow (p \vee q)$ and $out : \Pi p, q, r : Prop.(p \vee q) \rightarrow (p \rightarrow r) \rightarrow (q \rightarrow r) \rightarrow r$.

λ, Π and Σ. In order to facilitate manual input of pseudo terms we allow for the following abbreviations:

(2) SHORTHAND NOTATION:

```
[V1:PT1,..,Vn:PTn]=>PT  =  pi(V1:PT1,..(pi(Vn:PTn,PT)..)
[V1:PT1,..,Vn:PTn]/\PT  =  sigma(V1:PT1,..
                           (sigma(Vn:PTn,PT)..)
```

`PT -> PT = pi(V1:PT1,PT)`, provided `V1` does not occur in `PT`.
`PT & PT = sigma(V1:PT1,PT)`, provided `V1` does not occur in `PT`.

```
~ PT  =  pi(V1:PT,false).
PT1\/PT2 = pi(V1:(pi(V2:A,false)),PT2)
```

Pseudo contexts are PROLOG lists containing introductions which are of the form `A:PT` or `def(A,PT1):PT2`. Here, `A` is a PROLOG atom. In general, we represent bound PTS variables (bound by λ, Π or Σ) with PROLOG variables. All other PTS variables (unbound) are represented using PROLOG atoms. This allows us to use the build-in PROLOG unification for comparing pseudo-terms.

Before the user calls the predicate `prove`, it is possible to set a number of parameters, using `setval(+Param,+Value)`. The parameters and respective values in question are: 1. `(feedback,on/off)`: for error report and status reports of the proof process (search depth), 2. `(base_context,classic/falsum/...)`: for setting the base contexts that should be loaded, 3. `(maxtime,seconds)`: maximum time that the proof search should continue, 4. `(maxdepth,integer)`: maximum search depth at which the system should search for a proof. The proof search always proceeds using the consecutive-bounded depth-first strategy. 5. `(startdepth,integer)`: the depth at which the consecutive-bounded depth-first strategy starts searching for proofs. 6. Finally, there are two parameters for determining how long reduction should continue and what proportion of memory reduction is allowed to use (to avoid looping and memory overflow for terms that do not normalize):

```
(reduce_time_out,milliseconds),
(reduce_space_out,number in [0..1]).
```

When alligator is consulted or compiled, the default settings of all the aforementioned parameters are printed to the screen.

3.1 Architecture Overview

3.2 Main Components

Add base contexts Here the context provided by the user is extended with those base context which are selected by the `base_context` parameter. Often this is the base context `classic`, which enables us to reason with the double negation rule.

Remove shorthand notation The syntactic sugaring which is used for the benefit of the user, to make input of queries easier is removed.

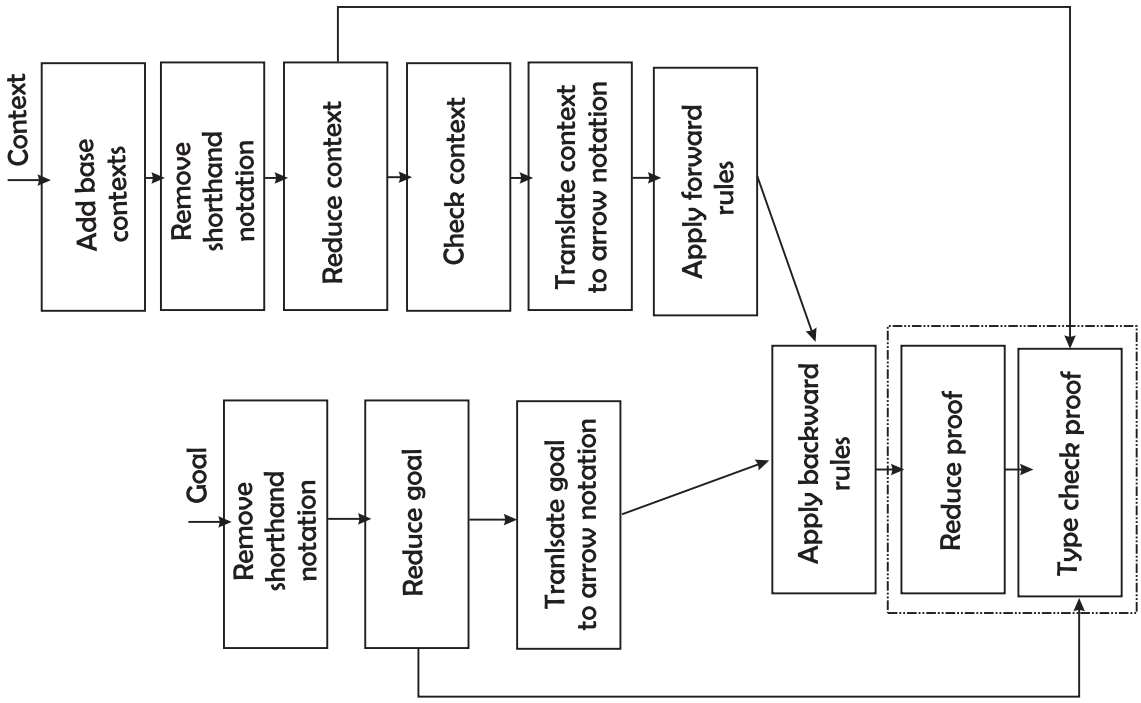

Figure 1: Overview of ALLIGATOR 1.0 Architecture

Reduce context/Goal In this step, all the types in the context and the goal type are normalized. This speeds up comparison of terms (which in PTS involves the conv rule) later on during proof construction. Note that if we are using one and the same context for proving several things, this step needs to be carried out only once, making the overall proof process more efficient.

Check context In the actual set-up of PTS, context checking and proof construction are interleaved. Basically, a context is checked as result of application of the start and weaken rule (e.g., when we reason backwards with the latter rule, we need verify that $\Gamma \vdash C : s$). For PTS, a legal context can therefore be defined as a context in which we can construct an inhabitant of some – it doesn't matter which – type. Because we separate context checking from proof construction, proof construction is more efficient and context checking does not need to be carried out for each new proof that we want to construct.

Translate to arrow notation This a flat representation format for pseudo-terms which is particularly suited for formulating forward and backward inference rules. In particular, in this format, we the notation for dependent function types is `[V1:PT1,..,Vn:PTn]=>PT`, instead of the recursive `pi(V1:PT1,..(pi(Vn:PTn,PT)..)`.

Apply forward rules We have three forward rules. Let us give two of these in schematic form (omitting proof objects, dependencies and recursion) and one in some more detail:

(3) a. `A => (B/\C)` $\longmapsto$ `A => B, A => C`

b. `A => (B => C)` $\longmapsto$ `[A,B] => C`

c. `P:sigma(X:A,B)` $\longmapsto$ `def(x1,pi1(P)):A` and
`def(x2,pi2(P)):B [X:=pi1(P)]`

Note that we use definitions for adding new information to the context that has been obtained through forward rules. A definition `def(X,Y):A` says that we can use the variable `X` as an abbreviation for the full proof `Y` of `A`.

Apply backward rules There are nine backward rules modelled on the PTS derivation rules. Let us look into the ALLIGATOR rule corresponding to (elim). For backwards reasoning, this is the most problematic rule. The reason for this is that when we go backwards in an attempt to prove $B[x := a]$, we need to guess an A such that $\Gamma \vdash F : (\Pi x : A.B)$ and $\Gamma \vdash a : A$. In the system, we circumvent this problem by attempting to find an `[X1:A1,...,Xn:An] => B` in the context. If that succeeds, the system adopts the new goals `A1,...,An`. There are many more details of the backwards rules (such as the use of membership instead of start and weaken, and the forward reasoning component of intro) which are beyond the scope of this paper, but which we hope to elucidate elsewhere, and which can also be learned from studying the source code of ALLIGATOR.

Although the performance of ALLIGATOR has been satisfactory on significant number of test input, it has to be noted that so far we have not formally verified the completeness of the system, in the sense that any proof that can be constructed in PTS$^{\Sigma}$ can also be constructed by ALLIGATOR.

Reduce proof and type check This is the part of our system which makes our prover particularly reliable. Independent of the proof construction, we check whether the proof that the system produced is actually a proper PTS$^{\Sigma}$ proof of our goal, given the input context. Fortunately, proof checking, as opposed to proof construction, is decidable. In principle, proof checking could also be done by a different program, since proof objects in PTS are in a standard format.

4 Proof Samples

In this section, we provide three examples of proofs that have been generated by ALLIGATOR, all took between $\frac{1}{10}$ and $\frac{1}{100}$ of a second to be constructed and type checked. We begin with two examples that show inferences which are relevant for anaphora resolution. We follow the approach to anaphora resolution detailed in [22]. In a nutshell, the idea is that an anaphoric expression triggers a proof goal that needs to be filled with a proof from the (local) context. Take, for instance, Partee's infamous bathroom sentence: 'Either there is no bathroom in this house, or it's in a funny place.'. We are interested in resolution of the pronoun 'it'. The disjunction $A \vee B_i$ (with i marking the anaphoric material) is mapped in our system to $(A \rightarrow \mathit{false}) \rightarrow B_i$. This means that we need to interpret B_i with respect to $\Gamma, A \rightarrow \mathit{false}$ (below, `a1` models the proof of this).

(4)
```
Goal = E:sigma(Y:e,q-Y)
Context = false:prop, e:set, q:[_444:e]=>prop,
a1:[_414:[_422:sigma(_429:e,q-_429)]=>false]=>false,
dn_pr:[_1348:prop,_1352:[_1360:[_1368:_1348]=>false]=>false]=>_1348
E = dn_pr-sigma(_429:e,q-_429)-a1
```

A second example models (we simplified the example by making it propositional) the inference behind the interpretation of 'It' in 'The barn contains a chain saw or a power drill. It ...' (page 205 of [18]). Here, we need to infer an object (something that is either a power drill or a chain saw) from a disjunction. Again we model disjunction in terms of negation and implication (see `a3` and `a1` and `a2` for the information that power drills and chain saws are both things (`u`)).

(5)
```
Goal Z:u
false:prop, b:prop, p:prop, q:prop, u:prop,
a1:[_470:p]=>u, a3:[_483:[_491:p]=>false]=>q,
a2:[_457:q]=>u,
dn_pr:[_1187:prop,_1191:[_1199:[_1207:_1187]=>false]=>false]=>_1187
Z = dn_pr-u-lambda(_1496:pi(_1523:u,false),_1496-(a2-
(a3-lambda(_1512:p,_1496-(a1-_1512)))))
```

Finally, the following example illustrates the inference from $\exists x \in e.\forall y \in e.r(x,y)$ to $\forall y \in e.\exists x \in e.r(x,y)$. The proof object in this example and the preceding one are also depicted in Figure 2.

(6)
```
Goal = G:[Y1:e]=>([X1:e]/\r-X1-Y1)
Context = false:prop, e:set,
f:sigma(_475:e,[_472:e]=>r-_475-_472),
dn_pr:[_1652:prop,_1656:[_1664:[_1672:_1652]=>false]=>false]=>_1652,
r:[_1696:e,_1691:e]=>prop,
def(v_13,pi1(f)):e,def(v_12,pi2(f)):[_1709:e]=>r-pi1(f)-_1709,
G = lambda(_978:e,pair(pi1(f),pi2(f)-_978))
```

5 Conclusions

In this paper we have argued that Dynamic Type Systems (DTS) are a useful formalism for computational semantics. We introduced a research tool, the ALLIGATOR theorem prover, which we hope will support further explorations of DTS in computational semantics. Needless to say that there are still many ways in which the implementation of ALLIGATOR can be improved and extended. To name a few: more extensive input error reporting (for context and type checking), extension of the test suite that comes with the system, efficient implementation of reasoning with equalities (currently, equality needs to be defined explicitly in the context), and extension of the system with the capability to undertake (a limited amount of) modal reasoning (see [8] for an extension of DTS with modalities).

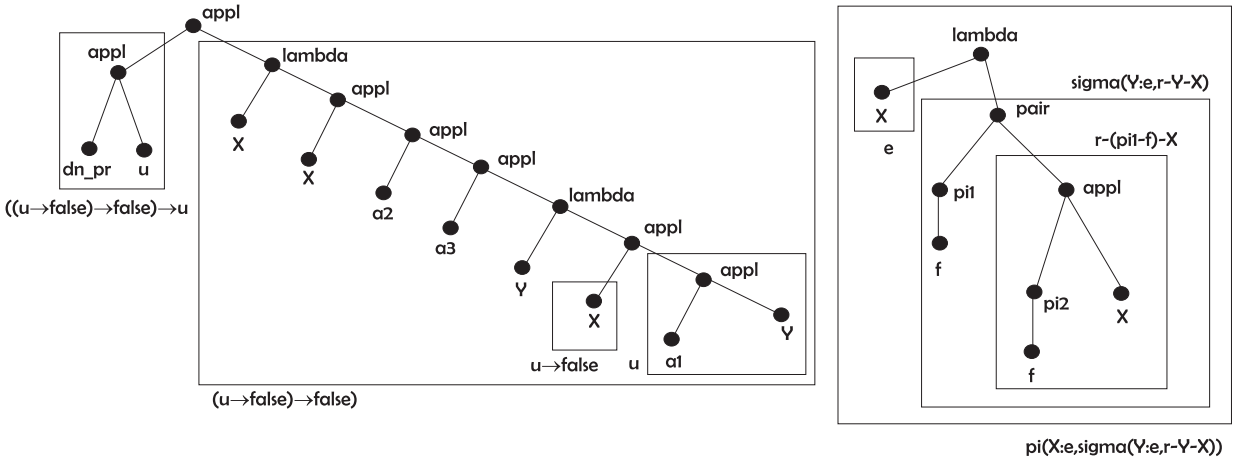

Figure 2: Representations of the (goal) proof objects in examples 5 and 6

References

[1] R. Ahn. *Agents, Objects and Events: A computational approach to knowledge, observation and communication*. PhD thesis, Eindhoven University of Technology, 2001.

[2] R. Ahn, R.J. Beun, T. Borghuis, H. Bunt, and C. van Overveld. The DenK-architecture: a fundamental approach to user-interfaces. *Artificial Intelligence Review Journal*, 8(2):431–445, 1994.

[3] P. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfennig, and H. Xi. TPS: A Theorem-Proving System for Classical Type Theory. *Journal of Automated Reasoning*, 16:321–353, 1996.

[4] H. Barendregt. Lambda Calculi with Types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Clarendon Press, Oxford, 1992.

[5] H. Barendregt and H. Geuvers. Proof-assistants using Dependent Type Systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2001.

[6] R.J. Beun, R.M. van Eijk, and H. Prüst. Ontological Feedback in Multiagent Systems. In N.R. Jennings, C. Sierra, L. Sonenberg, and M. Tambe, editors, *International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 110–117, New York, 2004. ACM Press.

[7] P. Blackburn and J. Bos. Computational Semantics. *Theoria*, 18(1):27–45, 2003.

[8] T. Borghuis. *Coming to Terms with Modal Logic: On the interpretation of modalities in typed λ-calculus*. PhD thesis, Eindhoven University of Technology, 1994.

[9] T. Borghuis and R. Nederpelt. Belief Revision with Explicit Justifications: An Exploration in Type Theory. CS-Report 00-17, Eindhoven University of Technology, 2000.

[10] H.C. Bunt and L.A. Kievit. Agent-dependent metonymy in a context-change model of communication. In H. Bunt, R. Muskens, and E. Thijsse, editors, *Computing Meaning II*, volume 77 of *Studies in Linguistics and Philosophy*, pages 75–95. Kluwer Academic Publishers, Dordrecht, 2001.

[11] R. Cooper and J. Ginzburg. Clarification ellipsis in dependent type theory. In J. Bos and C. Matheson, editors, *Proceedings of Edilog,the 6th Workshop on the Semantics and Pragmatics of Dialogue*. University of Edinburgh, September 2002.

[12] T. Fernando. A type reduction from proof-conditional to dynamic semantics. *Journal of Philosophical Logic*, (2):121–153, 2001.

[13] M. Franssen and H. de Swart. Cocktail: A Tool for Deriving Correct Programs. *Rev. R. Acad. Cien. Serie A. Mat.*, 98(1):95–111, 2004.

[14] G. Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 1935.

[15] H. Geuvers. *Logics and Type Systems*. PhD thesis, Catholic University of Nijmegen, 1993.

[16] H. Geuvers. A short and flexible proof of Strong Normalization for the Calculus of Constructions. In P. Dybjer, B. Nordström, and J. Smit, editors, *Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, pages 14–38, Berlin, 1994. Springer.

[17] G. Helman. *Restrictions on Lambda Abstraction and the Interpretation of Some Non-Classical Logics*. PhD thesis, University of Pittsburgh, 1977.

[18] H. Kamp and U. Reyle. *From Discourse to Logic: Introduction to Modeltheoretic Semantics for Natural Language, Formal Logic and Discourse Representation Theory*. Kluwer Academic Publishers, Dordrecht, 1993.

[19] R. Kempson, W. Meyer-Viol, and D. Gabbay. Syntactic Computation as Labelled Deduction: WH a case study. In R. Borsley and I. Roberts, editors, *Syntactic Categories*. Academic Press, 2000.

[20] T. Laan. *The Evolution of Type Theory in Logic and Mathematics*. PhD thesis, Eindhoven University of Technology, 1997.

[21] P. Piwek. ALLIGATOR Theorem Prover Home Page. mcs.open.ac.uk/pp2464/alligator, January 2006.

[22] P. Piwek and E. Krahmer. Presuppositions in Context: Constructing Bridges. In P. Bonzon, M. Cavalcanti, and R. Nossum, editors, *Formal Aspects of Context*, volume 20 of *Applied Logic Series*. Kluwer Academic Publishers, Dordrecht, 2000.

[23] A. Ranta. *Type-Theoretical Grammar*. Clarendon Press, 1994.

[24] L.J. Rips. *The Psychology of Proof: Deductive Reasoning in Human Thinking*. The MIT Press, Cambridge, Massachusetts, 1994.

[25] M. Stone. Specifying Generation of Referring Expressions by Example. In *Proceedings of AAAI Spring Symposium on Natural Language Generation in Spoken and Written Dialogue*, Stanford, March 2003.

[26] G. Sundholm. Proof Theory and Meaning. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume III, pages 471–506. D. Reidel, 1986.